All-Flash Storage
Corente Cloud Services Exchange
Developer Studio
Engineered Systems
Ksplice
NAS Storage
Netra Systems
Networking and Data Center Fabric Products
OpenStack
Oracle Linux
Oracle Optimized Solutions
Oracle VM
Oracle VM VirtualBox
Oracle x86 Servers
Private Cloud Appliance
SAN Storage
Secure Global Desktop
Server Management Tools
Software in Silicon
Software in Silicon Cloud
Solaris 10
Solaris 11
Solaris Cluster
SPARC Servers
StorageTek Tape Storage
Sun Blade 6000 Modular Systems

How to Use the Qualifier in C

By Douglas Walls, July 2003 (revised March 2006 and June 2016)

Using the `restrict` qualifier appropriately in C programs may allow the compiler to produce significantly faster executables.

Type Qualifiers

The C89 standards committee added two type qualifiers to C, `const` and `volatile`. The C99 committee added a third type qualifier with `restrict`. Individually, and in combination, these type qualifiers determine the assumptions a compiler makes when it accesses an object through an lvalue. The definition of an lvalue is an object locator, typically an identifier or the dereferencing of a pointer.

The syntax and semantics of `const` were adapted from C++; the concept itself has appeared in other languages. `volatile` and `restrict` are inventions of the committee; both follow the syntactic model of `const`.

Type qualifiers were introduced in part to provide greater control over optimization. Several important optimization techniques are based on the principle of cacheing: under certain circumstances the compiler can remember the last value accessed (read or written) from a location, and use this retained value the next time that location is read. (The memory, or cache, is typically a hardware register.) If this memory is a machine register, for instance, the code can be smaller and faster using the register rather than accessing external memory.

Type qualifiers can be characterized by the restrictions they impose on access and cacheing.

`const`
No writes through this lvalue. In the absence of this qualifier, writes may occur through this lvalue.

`volatile`
No cacheing through this lvalue: each operation in the abstract semantics must be performed (that is, no cacheing assumptions may be made, since the location is not guaranteed to contain any previous value). In the absence of this qualifier, the contents of the designated location may be assumed to be unchanged except for possible aliasing.

`restrict`
Objects referenced through a `restrict`-qualified pointer have a special association with that pointer. All references to that object must directly or indirectly use the value of this pointer. In the absence of this qualifier, other pointers can alias this object. Cacheing the value in an object designated through a `restrict`-qualified pointer is safe at the beginning of the block in which the pointer is declared, because no pre-existing aliases may also be used to reference that object. The cached value must be restored to the object by the end of the block, where pre-existing aliases again become available. New aliases may be formed within the block, but these must all depend on the value of the `restrict`-qualified pointer, so that they can be identified and adjusted to refer to the cached value. For a `restrict`-qualified pointer at file scope, the block is the body of each function in the file.

The `restrict` Qualifier and Aliasing

The `restrict` qualifier addresses the issue that potential aliasing can inhibit optimizations. Specifically, if a compiler cannot determine that two different pointers are being used to reference different objects, then it cannot apply optimizations such as maintaining the values of the objects in registers rather than in memory, or reordering loads and stores of these values.

The `restrict` qualifier was designed to express and extend two types of aliasing information already specified in the language.

First, if a single pointer is directly assigned the return value from an invocation of malloc, then that pointer is the sole initial means of access to the allocated object (that is, another pointer can gain access to that object only by being assigned a value that is based on the value of the first pointer). Declaring the pointer to be `restrict`-qualified expresses this information to the compiler. Furthermore, the qualifier can be used to extend a compiler's special treatment of such a pointer to more general situations. For example, an invocation of malloc might be hidden from the compiler in another function, or a single invocation of malloc might be used to allocate several objects, each referenced through its own pointer.

Second, the library specifies two versions of an object copying function, because on many systems a faster copy is possible if it is known that the source and target arrays do not overlap. The `restrict` qualifier can be used to express the restriction on overlap in a new prototype that is compatible with the original version:

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
void *memmove(void * s1, const void * s2, size_t n);
```

With the restriction visible to the compiler, a straightforward implementation of `memcpy` in C now gives a level of performance that previously required assembly language or other non-standard means. Thus the `restrict` qualifier provides a standard means with which to make, in the definition of any function, an aliasing assertion of a type that could previously be made only for library functions.

Some Typical Uses of the `restrict` Qualifier

The complexity of the specification of the `restrict` qualifier reflects the fact that C has a rich set of types and a dynamic notion of the type of an object. For example, an object does not have a fixed type, but acquires a type when referenced. Similarly, in some of the library functions, the extent of an array object referenced through a pointer parameter is dynamically determined, either by another parameter or by the contents of the array. The full specification is necessary to determine the precise meaning of a qualifier in any context, and so must be understood by the compiler. Fortunately, C programmers only need to understand a few simple patterns of usage explained in the following examples.

A compiler can assume that a file-scope `restrict`-qualified pointer is the sole initial means of access to an object, much as if it were the declared name of an array. This is useful for a dynamically allocated array whose size is not known until run time. Note in the following example how a single block of storage is effectively subdivided into two disjoint objects.

```
float * restrict a1, * restrict a2;
void init(int n)
{
  float * t = malloc(2 * n * sizeof(float));
  a1 = t; // a1 refers to 1st half
  a2 = t + n; // a2 refers to 2nd half
}
```

A compiler can assume that a `restrict`-qualified pointer that is a function parameter is, at the beginning of each execution of the function, the sole means of access to an object. Note that this assumption expires with the end of each execution. In the following example, parameters `a1` and `a2` can be assumed to refer to disjoint array objects because both are `restrict`-qualified. This implies that each iteration of the loop is independent of the others, and so the loop can be aggressively optimized.

```
void f1(int n, float * restrict a1, const float * restrict a2)
{
  int i;
  for ( i = 0; i < n; i++ )
  a1[i] += a2[i];
}
```

A compiler can assume that a `restrict`-qualified pointer declared with block scope is, during each execution of the block, the sole initial means of access to an object. An invocation of the macro shown in the following example is equivalent to an inline version of a call to the function `f1` above.

```
# define f2(N,A1,A2) \
{ int n = (N); \
 float * restrict a1 = (A1); \
 float * restrict a2 = (A2); \
 int i; \
 for ( i = 0; i < n; i++ ) \
  a1[i] += a2[i]; \
}
```

The `restrict` qualifier can be used in the declaration of a structure member. A compiler can assume, when an identifier is declared that provides a means of access to an object of that structure type, that the member provides the sole initial means of access to an object of the type specified in the member declaration. The duration of the assumption depends on the scope of the identifier, not on the scope of the declaration of the structure. Thus a compiler can assume that `s1.a1` and `s1.a2` below are used to refer to disjoint objects for the duration of the whole program, but that `s2.a1` and `s2.a2` are used to refer to disjoint objects only for the duration of each invocation of the `f3` function.

```
struct t {
  int n;
  float * restrict a1, * restrict a2;
};

struct t s1;

void f3(struct t s2) { /* ... */ }
```

Unions and Typedefs

The meaning of the `restrict` qualifier for a union member or in a type definition is analogous. Just as an object with a declared name can be aliased by an unqualified pointer, so can the object associated with a `restrict`-qualified pointer.

This allows the `restrict` qualifier to be introduced more easily into existing programs, and also allows `restrict` to be used in new programs that call functions from libraries that do not use the qualifier. In particular, a `restrict`-qualified pointer can be the actual argument for a function parameter that is unqualified. On the other hand, it is easier for a translator to find opportunities for optimization if as many as possible of the pointers in a program are `restrict`-qualified.

Further Information

Latest C Programming Language Standard
The 2011 ISO C standard is available for purchase and download in PDF from the ANSI Electronic Standars Store. Enter "9899" in the search field to find it. Be sure to also download the free Corrigendum of corrections to the standard, found with the same search.
Oracle Developer Studio 12.5 C User's Guide
Download the latest user's guide for the Oracle Developer Studio C Compiler.

About the Author

Douglas Walls is the C compiler project lead at Oracle, and is a member of JTC1/SC22/WG14 international standards committee that developed the ISO 1999 and 2011 C Standards.

E-mail this page Printer View

ORACLE CLOUD
Learn About Oracle Cloud Computing
Get a Free Trial
Learn About DaaS
Learn About SaaS
Learn About PaaS
Learn About IaaS

JAVA
Learn About Java
Download Java for Consumers
Download Java for Developers
Java Resources for Developers
Java Cloud Service
Java Magazine

CUSTOMERS AND EVENTS
Explore and Read Customer Stories
All Oracle Events
Oracle OpenWorld
JavaOne

EMAIL SUBSCRIPTIONS

COMMUNITIES
Blogs
Discussion Forums
Wikis
Oracle ACEs
User Groups
Social Media Channels

SERVICES AND STORE
Log In to My Oracle Support
Training and Certification
Become a Partner
Find a Partner Solution
Purchase from the Oracle Store
Source Code for Open Source Software